Multi-core Architecture and Programming

Yang Quansheng(杨全胜)
http://www.njyangqs.com/

School of Computer Science & Engineering
Southeast University

Programming with MPI

- Content
  - What is MPI
  - Basic of programming with MPI
  - Collective Communication

What is MPI

- MPI is a message-passing application programmer interface .
  - The MPI is a language-independent communications protocol used to program parallel computers.
  - MPI's goals are high performance, scalability, and portability.
  - It has become the *de facto* standard for communication among processes that model a parallel program running on a distributed memory system.

What is MPI

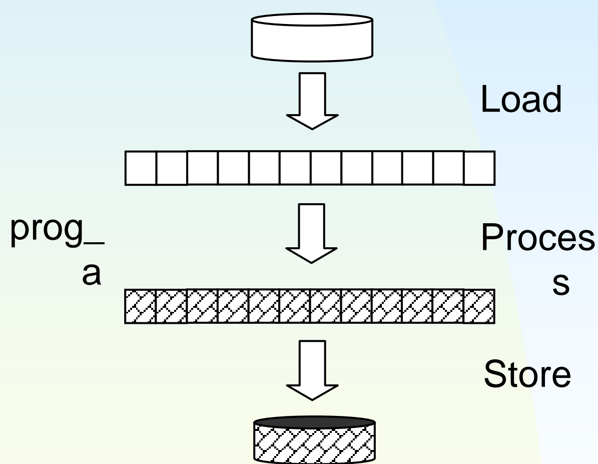
- MPI is a message-passing application programmer interface .
  - The principal MPI-1 model has no shared memory concept, and MPI-2 has only a limited distributed shared memory concept.
  - MPI includes point-to-point message passing and collective (global) operations, all scoped to a user-specified group of processes.
  - Every process has its own stack and code segment, data transfer among the processes can be done by calling the communication function explicitly.

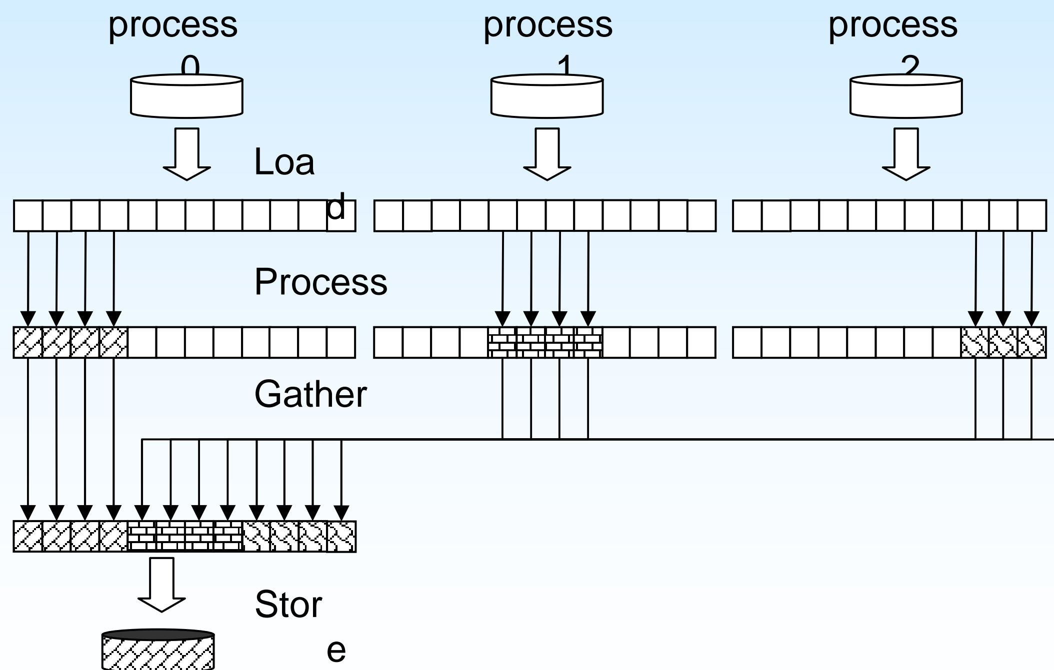
What is MPI

- MPI is a message-passing application programmer interface .
  - MPI was designed for high performance on both massively parallel machines and on workstation clusters
  - MPI is widely available, with both free available and vendor-supplied implementations
    - Open Source: MPICH, LAM MPI
    - Non Open Source: INTEL MPI

What is MPI

- Types of MPI program
  - SPMD (Single Program Multiple Data)

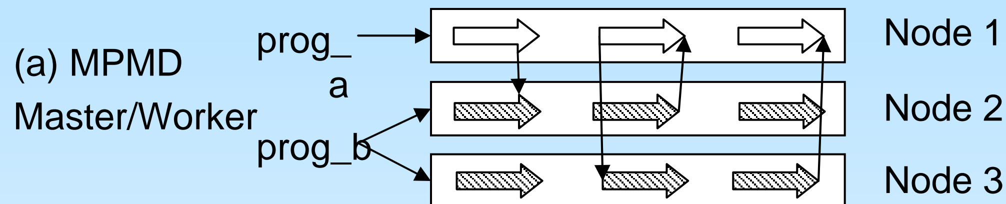




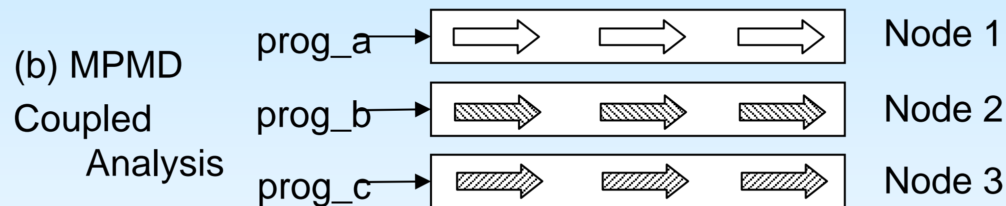
What is MPI

- Types of MPI program
  - MPMD (Multiple Program Multiple Data)

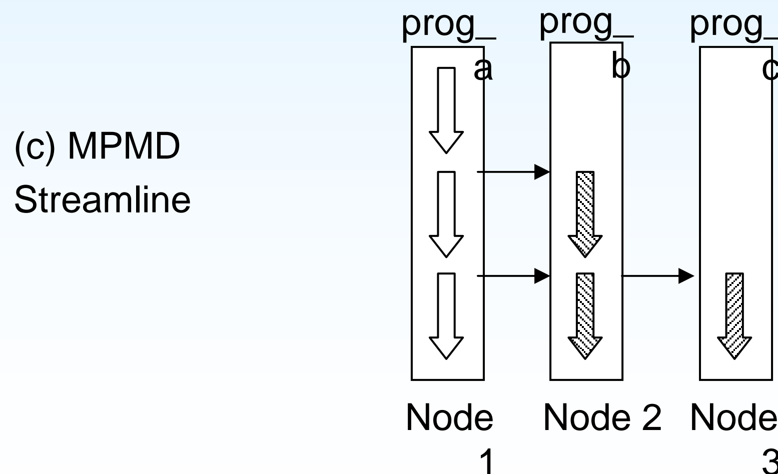
(a) MPMD Master/Worker



(b) MPMD Coupled Analysis



(c) MPMD Streamline



Programming with MPI

- Content
  - What is MPI
  - Basic of programming with MPI
  - Collective Communication

Basic of programming with MPI

- Four basic functions
  - int MPI_Init(int **argc*, char ****argv*);
    - MPI_Init initial the MPI execution environment.
  - int MPI_Finalize(void);
    - Terminates MPI execution environment.
  - int MPI_Comm_rank(MPI_Comm *comm*, int **rank*);
    - Determines the rank of the calling process in the communicator
  - int MPI_Comm_size(MPI_Comm *comm*, int **size*);
    - Determines the size of the group associated with a communicator

Basic of programming with MPI

- An Example

```
#include <stdio.h>
#include "mpi.h"

int main ( int argc, char *argv[] ) {
    int rank;
    int size;

  MPI_Init ( argc, argv ) ;
  MPI_Comm_rank (MPI_COMM_WORLD, &rank) ;
  MPI_Comm_size (MPI_COMM_WORLD, &size) ;
  printf ( "Hello world from process %d of %d\n", rank, size
) ;
  MPI_Finalize () ;
  return 0;
}
```

```
Hello world from process 0 of 4
Hello world from process 1 of 4
Hello world from process 2 of 4
Hello world from process 3 of 4
```

Basic of programming with MPI

- Point to Point Message Passing
  - int MPI_SEND(buf, count, datatype, dest, tag, comm)
    - Performs a blocking send to the *dest* process which in the *comm*。 The number of data in *buf* which will be send is *count*, the type of them is *datatype*.
  - int MPI_RECV(buf,count, datatype, source, tag, comm, status)
    - Blocking receive for a message.

Basic of programming with MPI

- Point to Point Message Passing
  - void *buf: initial address of send/receive buffer
  - int count: maximum number of elements in buffer
  - MPI_Datatype datatype: datatype of each buffer element
  - int dest/source: rank of destination/source
  - int tag: message tag
  - MPI_Comm comm: communicator
  - MPI_Status *status: status object

Basic of programming with MPI

- Error treatment
  - int MPI_Abort(MPI_Comm *comm*, int *errorcode*);
    - Terminates all MPI processes associated with the communicator comm; currently this function terminates *all* processes.
    - int errorcode: error code to return to invoking environment

```
#include <mpi.h>
int main(int argc, char *argv[])
{
    MPI_Init (NULL, NULL);
    MPI_Abort (MPI_COMM_WORLD, 911);
    /* No further code will execute */
    MPI_Finalize();
    return 0;
}
```

Programming with MPI

- Content
  - What is MPI
  - Basic of programming with MPI
  - Collective Communication

Collective Communication

- Synchronization
  - int MPI_Barrier(MPI_Comm *comm*);
    - Blocks the caller until all processes in the communicator have called it; that is, the call returns at any process only after all members of the communicator have entered the call.
    - This routine may be safely used by multiple threads without the need for any user-provided thread locks.
    - The routine is not interrupt safe.

Collective Communication

- Broadcast
  - int MPI_Bcast(void *buffer*, int *count*, MPI_Datatype *datatype*, int *root*, MPI_Comm *comm*);
    - Broadcasts a message from the process with rank root to all processes of the group, itself included.
    - This routine may be safely used by multiple threads without the need for any user-provided thread locks.
    - The routine is not interrupt safe.

Collective Communication

- Gather
  - int MPI_Gather(void **sendbuf*, int *sendcnt*, MPI_Datatype *sendtype*, void **recvbuf*, int *recvcnt*, MPI_Datatype *recvtype*, int *root*, MPI_Comm *comm*);
    - Each process (root process included) sends the contents of its send buffer to the root process.
    - The root process receives the messages and stores them in rank order.
    - This routine is thread-safe. However, the routine is not interrupt safe.

Collective Communication

- Scatter
  - int MPI_Scatter(void **sendbuf*, int *sendcnt*, MPI_Datatype *sendtype*, void **recvbuf*, int *recvcnt*, MPI_Datatype *recvtype*, int *root*, MPI_Comm *comm*);
    - Sends data from one process to all other processes in a communicator
    - is the inverse operation to MPI_GATHER
    - This routine is thread-safe. However, the routine is not interrupt safe.

Collective Communication

- All Gather
  - int MPI_Allgather(void **sendbuf*, int *sendcount*, MPI_Datatype *sendtype*, void **recvbuf*, int *recvcount*, MPI_Datatype *recvtype*, MPI_Comm *comm*);
    - Gathers data from all tasks and distribute the combined data to all tasks
    - MPI_ALLGATHER can be thought of as MPI_GATHER, but where all processes receive the result, instead of just the root.
    - This routine is thread-safe. However, the routine is not interrupt safe.

Collective Communication

- ALL to ALL
  - int MPI_Alltoall(void **sendbuf*, int *sendcount*, MPI_Datatype *sendtype*, void **recvbuf*, int *recvcount*, MPI_Datatype *recvtype*, MPI_Comm *comm*);
    - is an extension of MPI_ALLGATHER to the case where each process sends distinct data to each of the receivers instead of the same data.
    - This routine is thread-safe. However, the routine is not interrupt safe.

Data Scoping to Protect Data

- Data Scope Attributes
  - The default status can be modified with

```
default (shared | none)
```

  - Scoping attribute clauses

```
shared(varname,…)
```

```
private(varname,…)
```

Data Scoping to Protect Data

- The Private Clause
  - Reproduces the variable for each thread
    - Variables are un-initialized; C++ object is default constructed
    - Any value external to the parallel region is undefined

```
void* work(float* c, int N) {
  float x, y; int i;
 #pragma omp parallel for private(x,y)
     for(i=0; i<N; i++) {
             x = a[i]; y = b[i];
             c[i] = x + y;
     }
}
```

Data Scoping to Protect Data

- Example: Dot Product

```
float dot_prod(float* a, float* b, int N)
{
  float sum = 0.0;
#pragma omp parallel for shared(sum)
   for(int i=0; i<N; i++) {
     sum += a[i] * b[i];
   }
  return sum;
}
```

What is wrong?

Data Scoping to Protect Data

- Protect Shared Data
  - Must protect access to shared, modifiable data

```
float dot_prod(float* a, float* b, int N)
{
  float sum = 0.0;
#pragma omp parallel for shared(sum)
   for(int i=0; i<N; i++) {
#pragma omp critical
      sum += a[i] * b[i];
   }
  return sum;
}
```

Programming with OpenMP

- Content
  - What is PpenMP
  - Parallel Regions
  - Work-sharing Construct
  - Data scoping to Protect Data
  - Explicit Synchronization
  - Scheduling Clauses
  - Other helpful Construct and Clauses

Explicit Synchronization

- OpenMP* Critical Construct

```
#pragma omp critical [(lock_name)]
```

- Defines a critical region on a structured block

Threads wait their turn –at a time, only one calls `consum()` thereby protecting R1 and R2 from race conditions

Naming the critical construct is optional, but may increase performance

```
float R1,R2;
#pragma omp parallel
{ float B;
#pragma omp for
  for(int i=0; i<niters; i++){
    B = big_job(i);
#pragma omp critical (R1_lock)
    consum (B, &R1);
    A = bigger_job(i)
#pragma omp critical (R1_lock)
    consum (A, &R2);
  }
}
```

Explicit Synchronization

- OpenMP* Reduction Clause

```
reduction (op : list)
```

  - The variables in "*list*" must be shared in the enclosing parallel region
  - Inside parallel or work-sharing construct:
    - A PRIVATE copy of each list variable is created and initialized depending on the "op"
    - These copies are updated locally by threads
    - At end of construct, local copies are combined through "op" into a single value and combined with the value in the original SHARED variable

Explicit Synchronization

- Reduction Example

```
#pragma omp parallel for reduction(+:sum)
  for(i=0; i<N; i++) {
    sum += a[i] * b[i];
  }
```

- Local copy of *sum* for each thread
- All local copies of *sum* added together and stored in "global" variable

Explicit Synchronization

- C/C++ Reduction Operations
  - A range of associative operands can be used with reduction
  - Initial values are the ones that make sense mathematically

Operand	Initial Value
+	0
*	1
-	0
^	0

Operand	Initial Value
&	~0
|	0
&&	1
||	0

Programming with OpenMP

- Content
  - What is PpenMP
  - Parallel Regions
  - Work-sharing Construct
  - Data scoping to Protect Data
  - Explicit Synchronization
  - Scheduling Clauses
  - Other helpful Construct and Clauses

Scheduling Clauses

- Assigning Iterations
  - The Schedule clause affects how loop iterations are mapped onto threads

schedule(*static [, chunk]*)

- Blocks of iterations of size "chunk" to thread
- Round robin distribution

schedule(*dynamic [, chunk]*)

- Threads grab "chunk" iterations
- When done with iterations, thread request next set

schedule(*guided [, chunk]*)

- Dynamic schedule starting with large block
- Size of the blocks shrink; no smaller than "chunk"

Scheduling Clauses

- Which Schedule to Use

Schedule Clause	When To Use
STATIC	Predictable and similar work per iteration
DYNAMIC	Unpredictable, highly variable work per iteration
GUIDED	Special case of dynamic to reduce scheduling overhead

Scheduling Clauses

- Schedule Clauses Example

```
#pragma omp parallel for schedule(static, 8)
  for(I = start; I <= end; i+=2 ) {
    if (TestForPrime(i) ) gPrimesFound++;
  }
```

- Iterations are divided into chunks of 8
  - If start = 3, then first chunk is i={3,5,7,9,11,13,15,17}

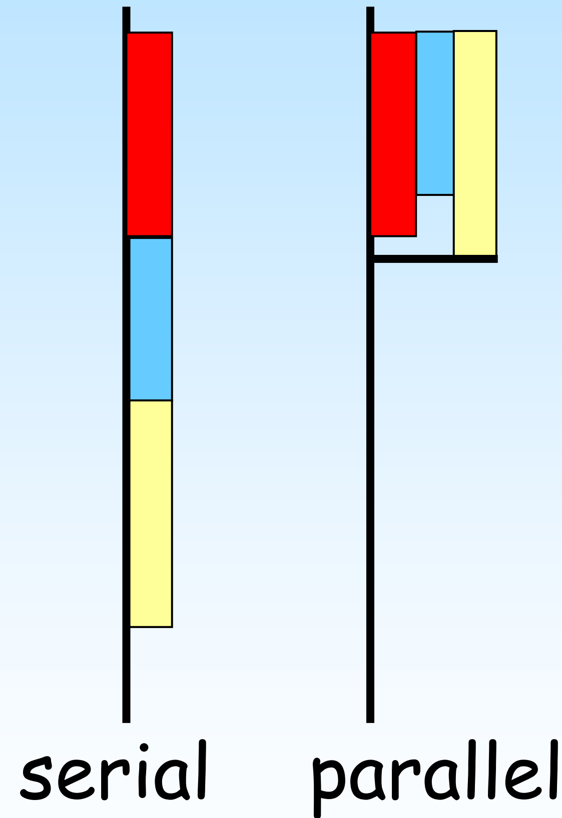
Programming with OpenMP

- Content
  - What is PpenMP
  - Parallel Regions
  - Work-sharing Construct
  - Data scoping to Protect Data
  - Explicit Synchronization
  - Scheduling Clauses
  - Other helpful Construct and Clauses

Other helpful Construct and Clauses

- Parallel Sections

```
#pragma omp parallel sections
{
    #pragma omp section
    phase1();
    #pragma omp section
    phase2();
    #pragma omp section
    phase3();
}
```



Other helpful Construct and Clauses

- Single Construct
  - Denotes block of code to be executed by only one thread
    - First thread to arrive is chosen
  - Implicit barrier at end

```
#pragma omp parallel
{
  DoManyThings();
#pragma omp single
  {
   ExchangeBoundaries();
  }  // threads wait here for single
  DoManyMoreThings();
}
```

Other helpful Construct and Clauses

- Master Construct
  - Denotes block of code to be executed only by the master thread
  - No implicit barrier at end

```
#pragma omp parallel
{
  DoManyThings();
#pragma omp master
  {                          // if not master skip to next stmt
    ExchangeBoundaries();
  }
  DoManyMoreThings();
}
```

Other helpful Construct and Clauses

- Implicit Barriers
  - Several OpenMP* constructs have implicit barriers
    - parallel
    - for
    - single
  - Unnecessary barriers hurt performance
    - Waiting threads accomplish no work!
  - Suppress implicit barriers, when safe, with the nowait clause

Other helpful Construct and Clauses

- Nowait Clause

```
#pragma omp for
nowait
   for(…)
     {…};
```

```
#pragma omp single  nowait
   {  […]  }
```

Use when threads would wait between independent computations

```
#pragma omp for  schedule (dynamic, 1) nowait
   for(int i=0; i<n; i++)
     a[i] = bigFunc1(i);

#pragma omp for  schedule (dynamic, 1)
   for(int j=0; j<m; j++)
     b[j] = bigFunc2(j);
```

Other helpful Construct and Clauses

- Barrier Construct
  - Explicit barrier synchronization
  - Each thread waits until all threads arrive

```
#pragma omp parallel shared (A, B, C)
{
        DoSomeWork(A,B);
        printf("Processed A into B\n");
#pragma omp barrier
        DoSomeWork(B,C);
        printf("Processed B into C\n");
}
```

Other helpful Construct and Clauses

- Atomic Construct
  - Special case of a critical section
  - Applies only to simple update of memory location

```
#pragma omp parallel for shared(x, y, index, n)
  for (i = 0; i < n; i++) {
    #pragma omp atomic
      x[index[i]] += work1(i);
    y[i] += work2(i);
  }
```

Other helpful Construct and Clauses

- OpenMP* API
  - Get the thread number within a team

```
int omp_get_thread_num(void)
```

  - Increment semaphore (Post operation)

```
int omp_get_num_thread (void)
```

  - Usually not needed for OpenMP codes
    - Can lead to code not being serially consistent
    - Does have specific uses (debugging)
    - Must include a header file

```
#include <omp.h>
```

Programming with OpenMP What's Been Covered

- OpenMP* is:
  - A simple approach to parallel programming for shared memory machines
- We explored basic OpenMP coding on how to:
  - Make code regions parallel (omp parallel)
  - Split up work (omp for)
  - Categorize variables (omp private….)
  - Synchronize (omp critical…)
- We reinforced fundamental OpenMP concepts through several labs

Advanced Concepts

More OpenMP*

- Data environment constructs
  - FIRSTPRIVATE
  - LASTPRIVATE
  - THREADPRIVATE

Firstprivate Clause

- Variables initialized from shared variable
- C++ objects are copy-constructed

```
incr=0;
#pragma omp parallel for firstprivate(incr)
for (I=0;I<=MAX;I++) {
        if ((I%2)==0) incr++;
        A(I)=incr;
}
```

Lastprivate Clause

- Variables update shared variable using value from last iteration
- C++ objects are updated as if by assignment

```
void sq2(int n, double *lastterm)
{
  double x; int i;
  #pragma omp parallel
  #pragma omp for lastprivate(x)
  for (i = 0; i < n; i++){
     x = a[i]*a[i] + b[i]*b[i];
     b[i] = sqrt(x);
  }
  lastterm = x;
}
```

Threadprivate Clause

- Preserves global scope for per-thread storage
- Legal for name-space-scope and file-scope
- Use copyin to initialize from master thread

```
struct Astruct A;
#pragma omp threadprivate(A)
…

#pragma omp parallel
   copyin(A)
  do_something_to(&A);
…

#pragma omp parallel
  do_something_else_to(&A);
```

Private copies of "A" persist between regions

Performance Issues

- Idle threads do no useful work
- Divide work among threads as evenly as possible
  - Threads should finish parallel tasks at same time
- Synchronization may be necessary
  - Minimize time waiting for protected resources

Load Imbalance

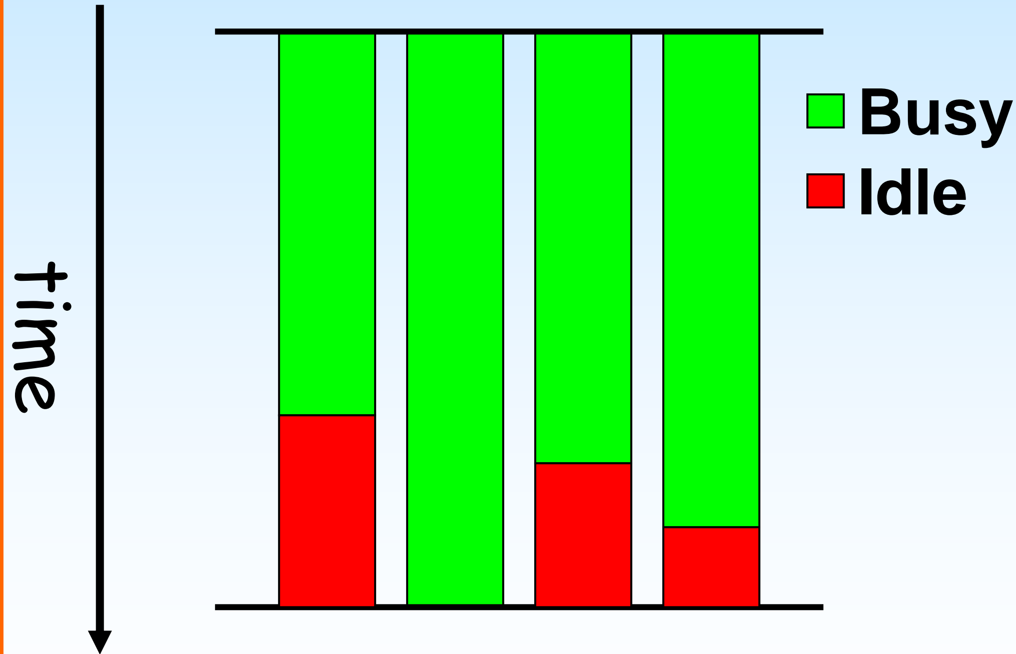
- Unequal work loads lead to idle threads and wasted time.

```
#pragma omp parallel
{

#pragma omp for
  for( ; ; ){

  }

}
```

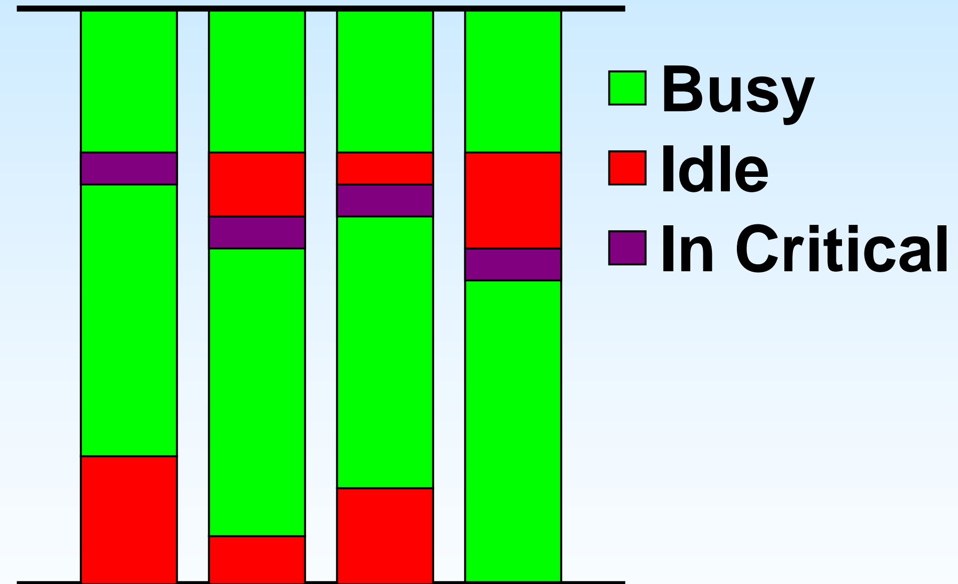


Synchronization

- Lost time waiting for locks

```
#pragma omp parallel
{

#pragma omp critical
  {
    ...
  }
  ...
}
```



Performance Tuning

- Profilers use sampling to provide performance data.
- Traditional profilers are of limited use for tuning OpenMP*:
  - Measure CPU time, not wall clock time
  - Do not report contention for synchronization objects
  - Cannot report load imbalance
  - Are unaware of OpenMP constructs

Programmers need profilers specifically designed for OpenMP.

Static Scheduling: Doing It By Hand

- Must know:
  - Number of threads (Nthrds)
  - Each thread ID number (id)
- Compute start and end iterations:

```
#pragma omp parallel
{
        int i, istart, iend;
        istart = id * N / Nthrds;
        iend = (id+1) * N / Nthrds;
        for(i=istart;i<iend;i++){
           c[i] = a[i] + b[i];}
}
```